Examen de Programmation Objet (Java)
2018–2019

— Durée : 2h
— Type : ExamManager
— Rendu : une archive contenant les sources Java et un fichier texte pour les réponses écrites, à déposer dans le dossier `rendu`
— Javadoc : `/usr/share/doc/openjdk-8-jdk/api/index.html`
— Tous documents autorisés.
— Toutes vos affaires (sacs, vestes, *etc.*) doivent être placées à l'avant de la salle.
— Aucun téléphone ne doit se trouver sur vous ou à proximité, même éteint.
— Les déplacements et les échanges ne sont pas autorisés.
— Aucune question ne peut être posée aux enseignants, posez des hypothèses en cas de doute.

Pensez à bien vous assurer du bon déroulement du rendu avant de quitter la salle, car toute archive incorrecte ou vide sera notée zéro.

Problème (20 points)

Lorsqu'un programme manipule des nombres, il peut être utile de connaître l'unité de ceux-ci. Ainsi, plutôt que de représenter une vitesse par un simple nombre de type `double`, on peut la représenter par ce nombre accompagné de son unité (*e.g.*, ms^{-1} pour mètres par seconde). Cela permet, entre autres, de détecter certains calculs invalides, comme l'addition d'une vitesse en ms^{-1} et d'une distance en m.

Nous proposons d'écrire une classe `NombreAvecUnite` représentant un nombre accompagné de son unité. Pour simplifier les choses, les préfixes servant à désigner les multiples des unités (*e.g.*, le k dans kg ou km) ne sont pas traités. De ce fait, deux unités avec des noms différents sont toujours considérées comme différentes, même dans le cas où elles sont en réalité liées comme kg et g. Cette classe doit pouvoir s'utiliser ainsi :

```
1 NombreAvecUnite pi = new NombreAvecUnite(Math.PI); // 3.1415...
2 NombreAvecUnite temps = new NombreAvecUnite(22, "s").additionner(
3                             new NombreAvecUnite(17, "s")); // 39s
4 Map<String, Integer> unite = new HashMap<>();
5 unite.put("s", -1);
6 unite.put("m", 1);
7 NombreAvecUnite vitesse = new NombreAvecUnite(10, unite); // 10ms-1
8 NombreAvecUnite distance = vitesse.multiplier(temps); // 390m
```

1. Créez une classe `NombreAvecUnite` qui modélise un nombre avec unité. L'attribut contenant le nombre est de type `double` et celui contenant l'unité est une table associant à chaque nom d'unité (une chaîne) l'exposant correspondant (un entier). Par exemple, l'unité ms^{-1} est représentée par la table associant la chaîne `"m"` à l'entier 1 et la chaîne `"s"` à l'entier -1. Les composants de l'unité sont stockés dans la table par ordre alphabétique de leur nom. Écrivez les trois constructeurs suivants : (3pts)
   (a) Le premier prend en paramètre un nombre et son unité, chacun du même type que l'attribut qui lui correspond, comme illustré à la ligne 7. Ce constructeur doit tolérer la présence d'exposants nuls dans la table représentant l'unité passée en paramètre, mais ne doit pas les stocker dans l'attribut correspondant.

(b) Le second prend en paramètre un nombre et un nom d'unité (de type `String`), et construit ce nombre avec l'unité en question élevée à la puissance 1, comme illustré aux lignes 2 et 3.

(c) Le troisième prend en paramètre uniquement un nombre et construit ce nombre sans unité (*i.e.*, un nombre « pur »), comme illustré à la ligne 1.

2. Ajoutez des accesseurs en lecture et écriture pour respectivement lire et modifier la valeur du nombre. (1pt)
3. Écrivez une méthode d'instance `additionner` qui retourne la somme du nombre avec unité à laquelle on l'applique et de celui passé en paramètre, comme illustré aux lignes 2 et 3. Cette méthode lève une exception non vérifiée de type `UnitesInvalidesException` (à créer) lorsque les unités des deux nombres ne sont pas égales. (2pts)
4. Écrivez une méthode d'instance `multiplier` qui retourne le produit du nombre avec unité à laquelle on l'applique et de celui passé en paramètre, comme illustré à la ligne 8. (2pts)
5. Redéfinissez la méthode `equals`. Deux nombres avec unité sont égaux si les deux nombres sont égaux et de même unité. (1pt)
6. Redéfinissez la méthode `toString` afin qu'elle retourne une représentation textuelle du nombre avec unité dont les composants de l'unité apparaissent par ordre alphabétique de leur nom, séparés du nombre et les uns des autres par un espace. De plus, le caractère `^` est utilisé pour représenter l'exponentiation, et l'exposant 1 ne doit jamais apparaître. Par exemple, pour le nombre 7.5, la méthode `toString` retourne la chaîne `"7.5"`, et pour le nombre $10ms^{-2}$, la chaîne `"10.0 m s^-2"`. (2pts)

Nous souhaitons maintenant réaliser un programme permettant d'enregistrer la participation d'athlètes à des épreuves sportives. Dans notre programme, la participation d'un athlète à une épreuve est modélisée *via* l'interface suivante :

```
public interface Participation {
  String getNomAthlete();
  NombreAvecUnite getResultat();
  void afficher();
}
```

Cette interface comporte 3 méthodes :

— `getNomAthlete()` retourne le nom de l'athlète ;

— `getResultat()` retourne le résultat obtenu par l'athlète à l'épreuve à laquelle il a participé ;

— `afficher()` affiche le nom de l'athlète et les détails de son résultat obtenu. [1]

Dans cet examen, nous considérons trois épreuves (mais d'autres pourraient être ajoutées) : le sprint, la course à obstacles et le saut en hauteur. Il y a donc trois sortes de participations possibles pour un athlète. Une participation au sprint et à la course à obstacles sont caractérisées par la distance à parcourir et le temps de course (en secondes) de l'athlète. Une participation à la course d'obstacles est en plus caractérisée par le nombre d'obstacles touchés par l'athlète. Le résultat obtenu par l'athlète pour une participation au sprint est son temps de course, alors qu'à une course à obstacles c'est son temps de course auquel s'ajoute une pénalité d'une seconde par obstacle touché. Une participation au saut en hauteur est, quant à elle, caractérisée par la hauteur franchie (en mètres) par l'athlète (*i.e.*, son résultat obtenu).

Un exemple de trace d'exécution du programme souhaité est donné ci-dessous :

```
Usain Bolt
Sprint 100.0 m : 9.44 s
Vitesse : 10.593220338983052 m s^-1
```

1. Le format d'affichage d'une participation d'un athlète à une épreuve doit correspondre à celui donné dans l'exemple de trace d'exécution.

```
Usain Bolt
Sprint 200.0 m : 19.19 s
Vitesse : 10.422094841063052 m s^-1

Aries Merritt
Course à obstacles 110.0 m : 12.8 s
Vitesse : 8.59375 m s^-1

Kevin Mayer
Course à obstacles 110.0 m : 15.75 s
Vitesse : 8.0 m s^-1
Pénalité : 2s

Javier Sotomayor
Saut en hauteur : 2.45 m
```

Pour répondre aux questions suivantes, vous devez éviter de dupliquer du code et utiliser le polymorphisme, sous peine d'être pénalisé.

7. Toutes les classes créées doivent appartenir au *package* `sport`. (0.5pt)
8. Écrivez le code Java des classes `ParticipationSprint`, `ParticipationCourseAObstacles` et `ParticipationSautEnHauteur` permettant de modéliser respectivement la participation d'un athlète à une épreuve de sprint, de courses à obstacles et de saut en hauteur. (5pts)
9. Modifiez la classe `ParticipationSautEnHauteur` afin de pouvoir compter le nombre total de participations à l'épreuve de saut en hauteur. (1pts)
10. Créez une classe `Competition` contenant une méthode de classe `afficherParticipations( List<Participation> l)` qui réalise successivement l'affichage de chacune des participations de la liste passée en paramètre. Ajoutez ensuite une méthode `main` qui crée un tableau de participations d'athlètes à des épreuves et appelle la méthode `afficherParticipations`, afin de réaliser l'affichage donné dans l'exemple de trace d'exécution. (1.5pts)
11. Pourquoi le code suivant ne compile-t-il pas ? Modifiez votre code pour corriger cela. (1pt)

```
void dopage(List<ParticipationSprint> liste) {
  ...
  Competition.afficherParticipations(liste);
}
```